Reducing Database Locking Contention Through Multi-version Concurrency

(Mohammad Sadoghi et al., VLDB’14)

Presenter: Yihe Huang
Outline

● Multi-version databases
● Indexing challenges in MVDBs
● ACID semantics and traditional concurrency control in DBMS
● kV-Indirection MVCC protocol
● Evaluation and discussions
Multi-version Database

Copy-on-write update semantics

- Deletion of a record saves a copy of the old data
- Updates are never in-place -- allocate new entries for updated records and swap in the new entry identifiers

Retains update history

- Desired in business scenarios
- Sometimes essential for regulatory compliant
Pros and Cons of MVDB

Pros:

● Always good to have history (git)

Cons:

● Performance?
  ○ Searching for old versions is potentially expensive
  ○ Records are essentially “migratable” -- a given record doesn’t have a constant physical location in the storage media
    ■ Causes problems when indexing
Indexing Challenges

Indexes store physical Row IDs

- Every index needs to be updated to reflect inserts/updates/deletes

Read-write conflicts:

- Index updates block readers, even if the reader doesn’t care about the latest update
- Readers can’t efficiently retrieve history versions without a special “temporal index”
ACID Semantics and Concurrency Control

What do we mean by “correct” (transactionality)?

● Atomic, Consistent, Isolated, Durable
● Snapshot
● Serializable
● Repeatable
● Weaker semantics

“Let your reads and writes choose their own destinations!”
“Handle conflicts, by not handling ’em!”
-- Prof. James Mickens
Traditional Concurrency Control in DBMS

(Single-version) 2-Phase Locking

**Execute (Phase 1):**
- read(x): rd_lock(x), read record
- write(x): wr_lock(x), update record in-place

**Commit (Phase 2):**
- Release all locks
Problem with 2PL

Scalability problem due to reader-writer locks

- Reads and writes block each other
- Doesn’t make sense for a very “old” read to block a recent write
- Makes even less sense in a multi-version database
  - Writes need not conflict with reads since they keep old versions around
kV-Indirection MVCC

All problems in computer science can be solved by another level of indirection.
-- Butler Lampson

- Why store physical Row IDs in indexes when records can get moved around?
  - Same as: why use physical addresses when physical pages can be moved around?
- Indirection requires translation: additional LID-RID mapping must be separately maintained
  - Each entry in the LID-RID mapping table contains 2 RIDs
    - uRID, for uncommitted value (mostly used by writes)
    - cRID, for reads looking for the latest committed value
kV-Indirection MVCC

Index 1
lid1

Index 2
lid2

LID-RID map

uRID  cRID

Main table
kV-Indirection MVCC

The LID-RID indirection solves two problems:

- No more excessive index updates when the physical location of a record gets moved
  - Just to update the LID-RID mapping
- While a write is in-progress but not committed, reads can still access the latest committed version of the record, by following cRID
kV-Indirection MVCC

CC protocol: basically 2PL with LIDs

- **read(LID):** set `rd_lock(LID)`, and follow `cRID` to get data
- **write(LID):** set `wr_lock(LID)`, allocate a new row with `uRID` and modify

At commit:

- Acquire `cl_lock(LID)` for all LIDs in the write set
- Install changes (update LID-RID mapping)
- Release all locks

*For weaker reads the read-locks can be released prior to commit time (or not acquiring read-locks at all)*
Optimization

Serializable reads still block writes in the design

Solution: Be optimistic

- **read(LID):** observe cRID and optimistically proceed at execution
- **Validate that cRID hasn’t changed by commit time**
- **write(LID) follows the same pessimistic protocol**
Optimization

The new commit protocol:

1. Validate cRID for each record in the read set
2. Acquire \texttt{cl\_lock(LID)} for each record in the write set
3. Transaction will commit at this point
4. Install changes/update LID-RID mapping
5. Release all locks

In the validation step (step 1), \texttt{rd\_lock(LID)} is acquired prior to fetching the latest cRID value from the LID-RID mapping

- Read locks are only held for the duration of commit
Eval

- Much better than single-version CC when there are contentions
- Negligible overhead for the additional indirection
Discussions

1. Different isolation levels
2. Locks vs. atomic counters
3. Is it safe to use cRIDs as versions (and avoid a global clock)?